CSE 374: Lecture 19

Software Specification, HW6

Stubbing

Unit testing looks at one component at a time

Provide ‘stubs’ to give just enough code for executing the desired unit.

After unit testing succeeds, proceed with integration testing (combining units) and system testing (the entire product).

Testing frameworks exist to make this easier: *explore and use them!*

- Instead of computing a function, use a small table of pre-encoded answers
- Return default answers that won’t mess up what you’re testing
- Don’t do things (e.g., print) that won’t be missed
- Use an easier/ slower algorithm
- Use an implementation of fixed size (an array instead of a list?)
- Test with hard coded input.

Eat your vegetables

- Make tests
  - Early
  - easy to run (e.g., a make target with an automatic diff against sample output)
  - that test interesting and well-understood properties
  - that are as well-written and documented as other code
- Write the tests first! (seems odd until you do it)
- Write much more code than the “assignment requires you turn-in”
- Manually or automatically compute test-inputs and right answers?
- Write regression tests and run on each version to ensure bugs do not creep in for stuff that “used to work”.

Homework 5

Idea:

Write source code for a tree

Write tests to make sure the tree does what you want

ONLY THEN

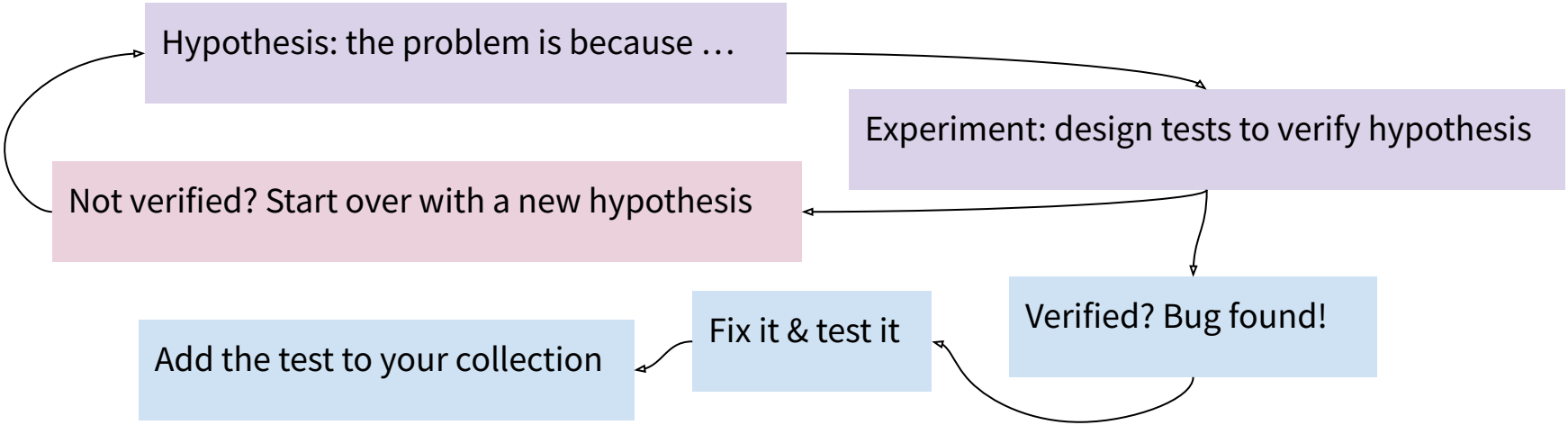
Write source code to use the tree as a trie.

What tests can you write?

- **Do letters become the correct number?**
- **Is memory allocated & deallocated correctly?**
  - **Hint: valgrind**

Testing to Debug

- Have bug? Find the cause and fix it
- A bit of an art, but, Treat debugging as a scientific experiment:



"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
Brian Kernighan -- Wrote THE BOOK on C (our book!)

Why Specs?

What is testing?

Software testing evaluates the **effectiveness** of a software sol

★ Systemat

★ Objective

But how do we know what it is supposed to do?

Effectiveness:

Does what it is supposed to do

Fails gracefully

Uses memory safely and efficiently

Computes in reasonable time

https://en.wikipedia.org/wiki/Software_testing

"Test your software or your users will."
Hunt & Thomas -- The Pragmatic Programmer

Full Specification

- Tractable for very simple stuff:
  - “Given integers *x*,*y>0*, return the greatest common divisor.”
- What about sorting a doubly-linked list?
  - Precondition: Can input be **`NULL`**? Can any **`prev`** and **`next`** fields be **`NULL`**? Can the list be circular or not?
  - Postcondition: Sorted (how to specify?, on what condition?)
- Beyond “pre” and “post” – time/space overhead, other effects (such as printing), things that happen in parallel
- Specs guide programming and testing!
- Declarative (“what” not “how”)
  - decouples implementation and use.

Basics: Pre and Post Conditions

- Pre- and post-conditions apply to any statement, not just functions
  - What is promised before and guaranteed after
- Because a loop “calls itself” its body’s post-condition better imply the loop’s precondition
  - A loop invariant
- CORRECT: a segment of code is correct if, when it begins execution in a state where its precondition is true, it is guaranteed to terminate in a state in which the postcondition is true
- Example: find max (next slide)

Find Max / Loop-invariant

```
// pre: arr has length
// len; len >= 1
int max = arr[0];
int i=1;
while(i<len) {
  if(arr[i] > max)
    max = arr[i];
  ++i;
}
// post: max >= all arr
// elements
```

loop-invariant: For all `j < i`, `max >= arr[j]`.

to show it holds after the loop body, must assume it holds before loop body

loop-invariant plus `!(i<len)` after body, enough to show post

Partial Specification

It may not be possible to completely specify an algorithm (or expedient)

Partial Specs:

- ➔ What is each argument precisely? Can arguments be null?
- ➔ Are pointers to stack data allowed? (what if stack is popped?)
- ➔ Are cycles in data structures allowed?
- ➔ Are there min and max sizes of data?

Checking specifications as part of code

- Specs are useful for more than writing code and testing
- Check them dynamically, e.g., with assertions
  - Easy: argument not `NULL`
  - Harder but doable: list not cyclic
  - Impossible: Does the caller have other pointers to this object?

Use ‘assert’ in C

```
#include <assert.h>
void f(int *x, int*y) {
  assert(x!=NULL);
  assert(x!=y);
  ...
}
```

- **assert** is a macro; ignore argument if **NDEBUG** defined at time of **#include**, else evaluate and if zero (false!) exit program with file/line number in error message
- Watch Out! Be sure that none of the code in an assert has side effects that alter the program’s behavior. Otherwise you get different results when assertions are enabled vs. when they are not

Remember this?

Unit Testing

Test small components of code individually

Basic approach - ‘assert’ desired performance.

(Note: Use conditional compilation
`Ifdef NDEBUG`
Plus macro
`#define assert(ignore)((void) 0)`
To compile without test code.)

```
#include <assert.h>
#include <stdlib.h>
#include "f.h"

// Assert statements will fail with a message
// if not true.
int main(int argc, char** argv) {

    assert(!f(0, 0));  // Test 1: f(0,0) => 0
    assert(f(0, 1));   // Test 2: f(0,1) => T
    assert(f(1, 0));   // Test 3: f(1,0) => T
    assert(f(1,1));    //  Test 4: f(1,1) => T

    // Test case 5: f(-1,1) => not-0
    assert(f(-1,1));
    return EXIT_SUCCESS;
}
```

```
OUTPUT >> program: f.c:9: main: Assertion
`!f(0,0)' failed.   Abort (core dumped)
```

Assert Style

- Often guidelines are simple and say “always” check everything, **but**:
  - Often not on “private” functions (caller already checked)
  - Unnecessary if checked statically
- Usually “Disabled” in released code because:
  - executing them takes time
  - failures are not fixable by users anyway
  - assertions themselves could have bugs/vulnerabilities
- Others say:
  - Should leave enabled; corrupting data on real runs is worse than when debugging

Exceptions

- Assert is used to verify internal expectations in code controlled by user
  - If asserts are violated code can be modified
- Exceptions are used to check expectations of code outside your control
  - Such as the return of a library function
  - Should usually exit (EXIT_FAILURE)
- Language dependent - Java offers asserts on top of its exception handling, C does not offer exception handling.
  - User is expected to anticipate trouble and catch it
  - Returning success/failure codes can be very helpful
- *Other Language dependent tools exist*
  - *Example: strong type checking prevents some sorts of specification violations*

API: Application Programming Interface

- Defines input and output for ‘applications’
  - Can be entire apps, or subfunctions, or classes
  - Library APIs describe available functions in library
- Useful for writing & testing
  - API dictates function prototype
  - (Black box?) Tests that show API adherence

Javadocs: Great example of an API standard

@param
@returns
@throws
@see
@author

Scientific Computing

Notes: worth specifying units in the function description and perhaps argument names.